# ED-cPSD Documentation

Andre Adam, Guang Yang, Huazhen Fang, Xianglin Li

Last Updated: March 7, 2025

# Contents

# Introduction

This document serves as reference and documentation for ED-cPSD, a software based on a novel algorithm of sequential erosion-dilation for continuous phase-size distribution (ED-cPSD) estimation. This document is segmented into a few different chapters which serve as reference for both users and contributors.

Chapter 1 is dedicated to get new users running the software and generating meaningful results immediately. Chapter 1 will walk new users through the installation process, basic GUI usage, running code de-coupled from the GUI, preparing inputs with ImageJ [1] and/or simple python scripts (OpenCV), expected outputs, and interpreting outputs.

Chapter 2 will first present an in-depth mathematical description to the algorithm working behind the scenes to calculate the continuous phase-size distribution (cPSD). This chapter also includes implementation details for both the standalone code and the GUI computational model. These implementation details are important for someone who is looking to contribute to and/or fork this project.

Chapter 3 presents several examples and case studies of the use of this software/algorithm. Each case study carries highlights quintessential aspects of the algorithm, such as discussions on how to interpret the output data given different inputs, comparisons with other cPSD and discrete phase-size distribution (dPSD) algorithms, and use cases in materials science and energy research. As more research involving this algorithm is published, the case-study section will grow accordingly.

Finally, chapter 4 has additional information, such as acknowledgments, code licensing information, and references.

# Chapter 1

# Quick-Start Guide

This chapter will be a brief introduction on how to download the software GUI and produce results with minimal setup required. This chapter will also mostly focus on Windows users. Some special instructions for Linux users are included in a separate section.

## 1.1 Setup

### 1.1.1 Method 1: Download Executable

The first step for getting the software running is to download it directly from one of the source repositories. For users that are inexperienced with Git, it is recommended to go directly to the project GitHub repository at `https://github.com/adama-wzr/ED_cPSD`. From there, select the green "Code" button, and select the option "Download Zip", as shown in figure 1.1.

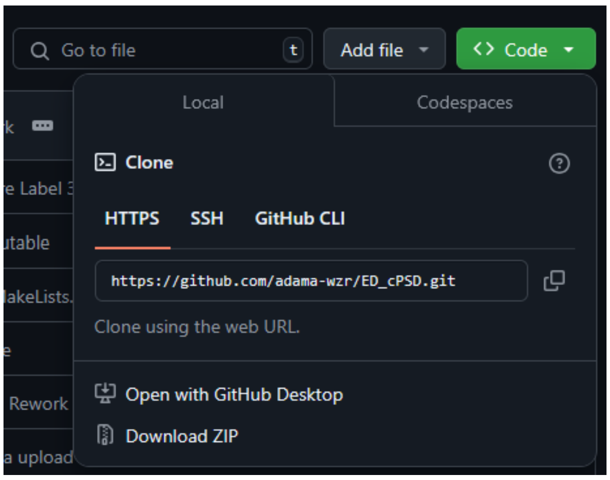


Figure 1.1: Select the green button, and then select the option to download the zip containing all files.

Once the zip is downloaded, it can be extracted to the desired location in the computer. The executable can be found in the folder "ED_cPSD/build/Desktop_Qt_6_8_1_MinGW_64_bit/". The executable is titled "ED_cPSD.exe". You can right click the ".exe" and create a shortcut, and then copy the shortcut to a place where it is more accessible, or just search for the executable using Windows search.

Verify at this stage that you are able to open the software. If you can open it, the everything should be running. Skip to one of the examples to learn how to use the software.

### 1.1.2 Method 2: Build Project

Statically linking libraries is relatively memory consuming on a computer, and might not be the best way to build the software for the experienced user. Thus, some may instead choose to build the executable.

In order to do that, just make sure the following requirements are met:

- Download the “GUI_src” folder from GitHub.
- Ensure a modern C++ version is installed.
- Ensure you have Qt6.8 installed and it is in the PATH (any recent major release of Qt should work, but only 6.8 was tested).
- (optional) Qt Creator 15.0 or newer (other versions might work).

Now, on the Qt creator, select the option to “Open Project”, navigate to the folder where the “GUI_src” was saved, and select the file “CMakeLists.txt”. A window will then appear asking the user to configure the project, but no additional configurations are need. Simply select the “Configure Project” button on the right side of this window, and Qt will build it for you.

At this stage, the GUI should be ready to run within the Qt Creator, simply press “Run” or “Ctrl-R”. The software outside of the Qt Creator if Qt is already in the PATH environment variables. A simpler alternative is to use the Qt Deployment Tool for Windows `https://doc.qt.io/qt-6/windows-deployment.html`, which will statically link the needed libraries from Qt directly to the executable file’s folder.

## 1.2 Examples

Also in the code repository, there is a folder labeled “Examples”, with three simulation examples (one in 2D and two in 3D). In this folder, the input files and expected outputs are already included, but in this section we will examine how to reproduce those examples.

Make sure the source files containing the structure information (`.csv` and `.jpg`) are saved at your computer. The input files and simulations will be run using the GUI, and that can be installed anywhere on the computer.

### 1.2.1 2D Example

This example will show you how to process an image using ImageJ and properly format it to be an acceptable input to the code. The image in question is `cRec2000.jpg`, a $2000 \times 2000$ reconstruction of a carbon electrode, featured in Adam 2022 [2].

The first step is to open the image using ImageJ. If the image is not of `.jpg`, the **FIRST** step is to go to the “File” tab, select “Save As”, and then save as `.jpg`.

Once the image is saved as `.jpg`, go to the “Image” tab, select the option “Type”, and make sure the image is saved as a “8-bit”. If there is a need for thresholding, go to the “Image” tab, “Adjust”, and then “Threshold”. Here the threshold can be adjusted. The software always assumes that white is pore, and black is particle. The image can either be converted to binary here, or the thresholding value to separate particle from pore can be entered into ED-cPSD directly. Remember to save again after these modifications are done.

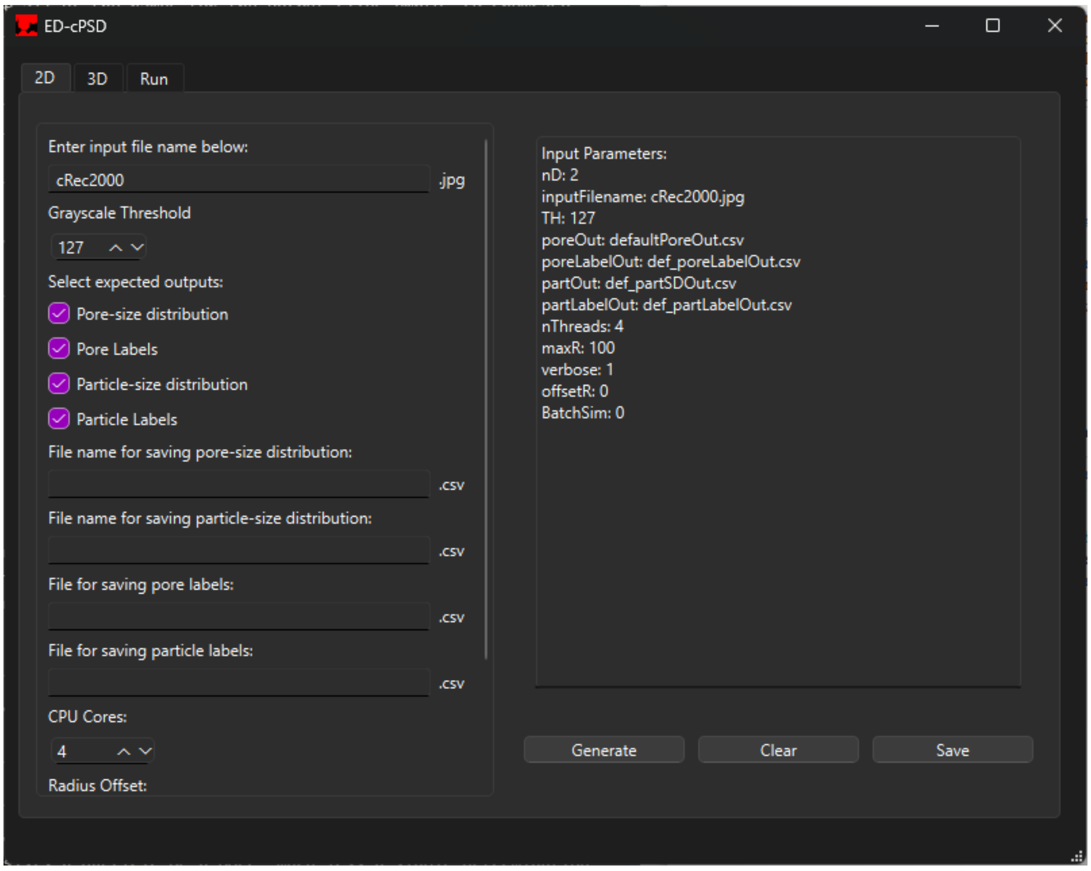
Once the image is saved, you can check that the process worked by right clicking on it, select “Properties”, then “Details”, and finally just verify that under bit-depth it says 8-bit. That indicates a single channel grayscale image. An RGB image will usually say 24 or 32-bit, and we don’t want that.

Now, we have to generate the input file for the simulation. Open the ED_cPSD software, and navigate to the 2D tab if necessary. Under the 2D tab, the very first field asks for the image name, which in this example is “cRec2000”. Do not enter the file extension, `.jpg`, as that is already entered automatically.

Beneath the file name, the threshold can be adjusted on a range from 1 to 255. Below that, the user can select the desired outputs. For the sake of the example, we will select all of the options. Also, we will leave all of the names for the output files empty, to showcase that defaults will be set in place if a user doesn’t select a name.

The last options relate to simulation parameters. The number of threads simply controls the number of CPU threads OpenMP will pool for the simulation. The radius offset can be used for smoothing, it means the simulation will start at a radius of $r = r_{\text{off}}$ instead of $r = 1$. The “Max. Radius” option sets a maximum radius cutoff for the simulations. Even if the solution is not yet complete, ED-cPSD will not scan beyond $r = r_{\text{max}}$. Finally, the option “Verbose” controls whether output will be printed to the window (GUI) or command line (for running the code on the command line).

Press the “Generate” button, and the input file will be displayed on the screen as seen in figure 1.2.



Enter input file name below:
cRec2000 .jpg
Grayscale Threshold
127
Select expected outputs:
✓ Pore-size distribution
✓ Pore Labels
✓ Particle-size distribution
✓ Particle Labels
File name for saving pore-size distribution:
.csv
File name for saving particle-size distribution:
.csv
File for saving pore labels:
.csv
File for saving particle labels:
.csv
CPU Cores:
4
Radius Offset:

```
Input Parameters:
nD: 2
inputFilename: cRec2000.jpg
TH: 127
poreOut: defaultPoreOut.csv
poreLabelOut: def_poreLabelOut.csv
partOut: def_partSDOut.csv
partLabelOut: def_partLabelOut.csv
nThreads: 4
maxR: 100
verbose: 1
offsetR: 0
BatchSim: 0
```

Figure 1.2: Example input file on ED-cPSD for the 2D example.

Press the "save" button, and be sure to save the input in the same folder as the image you are trying to simulate. Now go over to the "Run" tab, select the same folder again, or manually enter the path to the folder. At this stage, the simulation is ready to run, so just press the "Run" button and watch the results being printed to the text tab on the right-hand side of the screen.

Even though the image is $2000 \times 2000$, both pore and particle size distributions will be obtained within a few seconds. The GUI has a 0.25 $ms$ delay every time it prints something to avoid pointers being lost to garbage collection, thus it may take longer to print the output than it does to actually run the simulation. If this is an issue, consider turning "verbose" off or running the code in the command line directly (no delay).

For processing the output files, see some suggestions in section 1.3, where the topic is covered. It is also important to understand the assumptions of the model and what is being calculated, and this information can be found in chapter 2.

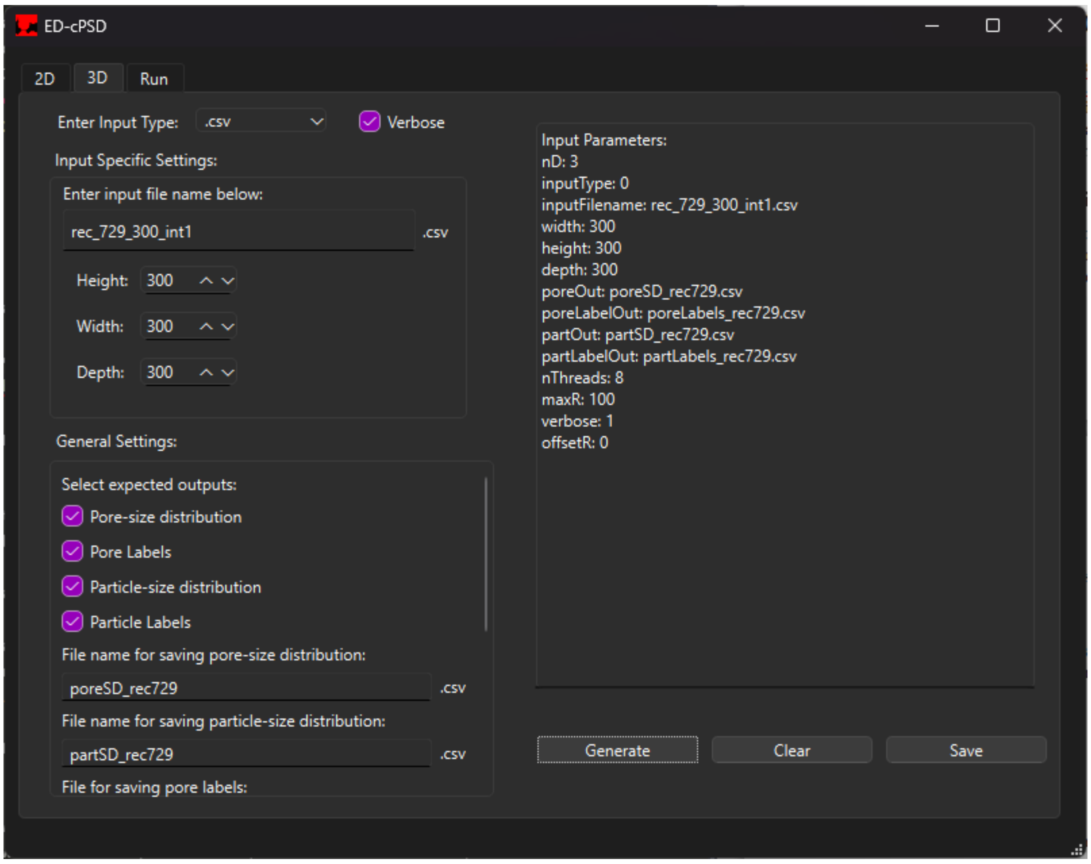
### 1.2.2 3D Example: csv Input

In this tutorial, we will cover another type of input: the `.csv` file. In this type of input, by convention, the `.csv` file's columns are the x, y, and z coordinates of all the solid particles in the domain. This type of input is not the most memory efficient in terms of memory storage, but it is easy and efficient to read in terms of programming, while also being easy to read as an output for either Paraview or Python scripts.

Open the ED-cPSD application, and open the "3D" tab. Under input type, change it from "Stack `.jpg`" to ".csv". The name of the input file provided under the "Example 3D csv" folder is "rec_729_300_int1.csv", which is a reconstruction of a carbon electrode featured in Adam 2022 [2]. This structure contains 300 voxels in each direction, a total of $2.7 \cdot 10^7$ voxels. The porosity is approximately 72.9%.

From this type of input, the code can't exactly obtain the size, so istead the user has to enter the dimensions in number of pixels. For this example, height, width, and depth are all 300. We can also select all of the possible outputs, and give names to the expected output files.

The last three options are the same as the 2D case. This time, we will increase the number of threads to 8 to display the parallel computing capabilities of ED-cPSD. Click "Generate" to see the a preview of the input file printed to the right-hand side panel on the screen. The window should look something like figure 1.3.



Enter Input Type: .csv
Verbose ✓

Input Specific Settings:

Enter input file name below:
rec_729_300_int1 .csv

Height: 300
Width: 300
Depth: 300

General Settings:

Select expected outputs:
✓ Pore-size distribution
✓ Pore Labels
✓ Particle-size distribution
✓ Particle Labels

File name for saving pore-size distribution:
poreSD_rec729 .csv

File name for saving particle-size distribution:
partSD_rec729 .csv

File for saving pore labels:

```
Input Parameters:
nD: 3
inputType: 0
inputFilename: rec_729_300_int1.csv
width: 300
height: 300
depth: 300
poreOut: poreSD_rec729.csv
poreLabelOut: poreLabels_rec729.csv
partOut: partSD_rec729.csv
partLabelOut: partLabels_rec729.csv
nThreads: 8
maxR: 100
verbose: 1
offsetR: 0
```

Figure 1.3: Example input file on ED-cPSD for the 3D example.

Save the input file on the same folder as the 3D structure `.csv` file. Switch over to the "Run" tab on the ED-cPSD software, and select the same folder again. Now press the "Run" button and wait for the simulation to run. You should see the outputs being produced and saved in the same folder as the original structure and the input file. For output processing, refer to section 1.3.

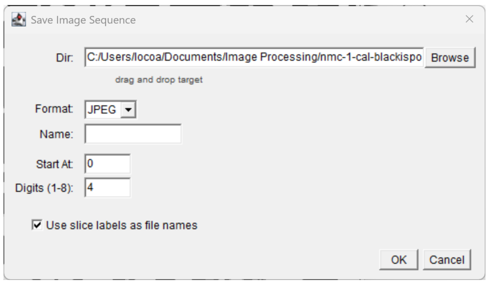
### 1.2.3 3D Example: Stack of .jpg

For this example, I will not provide a file. Instead, let's use a file from an open-source dataset as the example. Navigate to the NREL Battery Microstructure Library `https://www.nrel.gov/transportation/microstructure.html`, select the option to download samples. While there are several download options, I will use the segmented option for NMC-1-CAL where 1 denotes the active material, and 0 is everything else.

Once the `.tif` is downloaded, open it using ImageJ. The software will correctly interpret the file as a sequence of 2D images, so now it is on us to segment it properly and save the inputs.

While the image is already binary, the lowest bit depth in most conventional file formats is 8-bit. Thus, we will use ImageJ to further segment the image from $0 \rightarrow 1$ to $0 \rightarrow 255$. To do this, go under "Image", then "Adjust", and then "Threshold". Make sure this applied to all images in the stack.

Create a folder where you want to save the image slices. Then, on ImageJ, navigate to "File", then "Save As" and "Image Sequence". This will open a new dialog box with several options. Select the folder you just created, select the image format as "JPEG", and delete the image name. The default options for "Star at" and "Digits" should be good enough, and then make sure to check the box "use slice labels as file names". The window should look like figure 1.4.



Dir: C:/Users/locoa/Documents/Image Processing/nmc-1-cal-blackispo

Format: JPEG

Name:

Start At: 0

Digits (1-8): 4

☑ Use slice labels as file names

Figure 1.4: Screenshot of what the “Save Image Sequence” should look like on ImageJ side.

After saving, make sure under “Image” and “Type” it says “8-bit”. If some other option is ticked, make sure to change it to 8-bit and re-save the stack as `.jpg`.

If all of the aforementioned steps were followed, we should have a folder with a stack of `.jpg` files. The files have 4 digits with leading zeroes, and the first file is named “0000.jpg”. Now, on the ED-cPSD software, under the 3D tab, we can simply select the input type “Stack .jpg”.

The stack size is the number of images in the stack (in this case it is 320), and the number of leading zeroes is the same as “Digits” from ImageJ, figure 1.4. The other options are still the same from the 2D and 3D case.

Press “Generate”, and then save the input file in the same folder as the figures. Now you should be able to just go to the “Run” tab, select the folder again, and press the run button, and the simulation will begin.

## 1.3 Processing Output

The phase-size distributions are interpreted the same way whether they are in 2D or 3D, and this is addressed in subsection 1.3.1. The subsection contains information of how to transform the output to a size distribution based on diameter, include pixel resolutions, and how to get the average diameter, $\mathrm{D}_{50}$.

However, plotting the particle labels by radius can be tricky in 3D, and therefore that discussion is divided into a 2D discussion in subsection 1.3.2 and 3D in subsection 1.3.3.

### 1.3.1 Phase-Size Distribution Output

Regardless if the data was originally in 2D or in 3D, the format of this output is always the same. The first column will have the pore radius, with the label “r”, and the second column will have the probability density of a given particle of size “r”, and this column is labeled “p(r)”.

The first column can easily be converted to the real diameter, by using equation 1.1.

$$d = 2 \cdot r \cdot pr \tag{1.1}$$

where $pr$ is the pixel resolution of the base image.

An average diameter can be obtained from that in two ways. The simplest way to obtain $\mathrm{D}_{50}$ is by equation 1.2

$$\mathrm{D}_{50} = \sum_{d_{min}}^{d_{max}} d \cdot p\left(d\right) \tag{1.2}$$

Another way to obtain it is to calculate a cumulative probability distribution, as is done in other articles [3, 4]. This method interpolates and finds at what particle diameter the cumulative probability distribution reaches 0.5. These two methods arrive at the same $\mathrm{D}_{50}$.

### 1.3.2 Phase-Labels in 2D

The ED-cPSD label files have 4 columns in 2D, and the columns are the x and y pixel coordinates, the radius label “R”, and the label “L”, respectively. For more information on how “R” and “L” are derived, see section 2.4.

This brief tutorial will show how to plot a colormap of “R” on the original image domain. The output can be very informative for visualizing the approximate locations and concentrations of certain particle sizes and shapes. This also provides good understanding of how the the ED-cPSD code is measuring objects.

In order to plot, any plotting algorithm can be used, but for the sake of the tutorial, we will use Python with the libraries `matplotlib`, `pandas`, and `numpy`. The file used in this tutorial is included in the GitHub folder, under `/Examples/Contour Plot 2D`, with all the necessary files also included. The code below is well commented, and very straight-forward.

The code below produces figure 1.5 using the data from subsection 1.2.1.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

plt.rcParams["font.family"] = "Times New Roman"

# image size properties
xSize = 2000
ySize = 2000
pixel_Resolution = 1  # units

# Read image
imgName = "cRec2000.jpg"
img = np.uint8(mpimg.imread(imgName))

# Read .csv
df1 = pd.read_csv("def_partLabelOut.csv")

# get raw data from csv files
raw_data = df1[["x", "y", "R", "L"]].to_numpy()
R = np.zeros((ySize,xSize))

x = raw_data[:,0]
y = raw_data[:,1]

for i in range(len(raw_data[:,0])):
 R[y[i]][x[i]] = raw_data[i,2]*pixel_Resolution

# Create the mesh grid
Xp, Yp = np.meshgrid(np.linspace(0, 1, xSize), np.linspace(1.0*ySize/xSize, 0, ySize))

# plotting
fig1, ((ax1, ax2)) = plt.subplots(1, 2, constrained_layout=True)
fig1.set_dpi(100)
fig1.set_size_inches(8, 4)

# First axis is just the image
ax1.imshow(img)
ax1.set_title(imgName, fontsize=16)

# Second axis is R - contour
CS2 = ax2.contourf(Xp, Yp, R, 40, cmap=plt.cm.rainbow)
cbar2 = fig1.colorbar(CS2, ax=ax2, fraction=0.046, pad=0.04)
cbar2.set_label(r'Particle Radius [voxels]', rotation=90, fontsize=14)
ax2.set_title("Particle Radius Distribution", fontsize=16)
ax2.set_aspect('equal', adjustable='box')
plt.show()
```

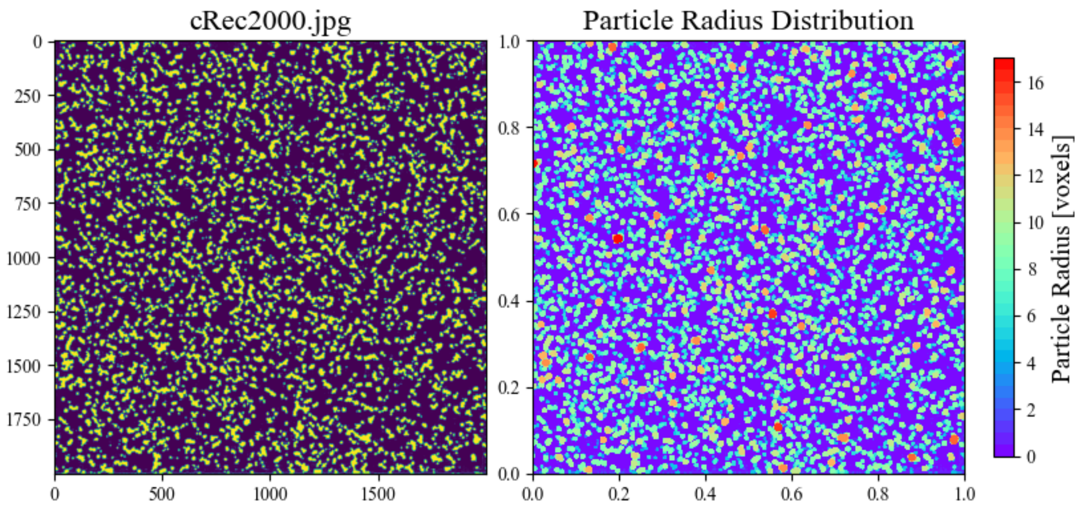


Figure 1.5: Example contour plot from processed ED-cPSD output.

### 1.3.3 Phase Labels in 3D

In this brief tutorial I will show another way of displaying the phase labels like in subsection 1.3.2. This can still be done using Python, but instead this tutorial will display another technique using the open-source software Paraview [5].

The ED-cPSD label files have 5 columns in 3D, and the columns are the x, y, and z pixel coordinates, the radius label "R", and the label "L", respectively. The output `.csv` file can be read directly into Paraview. Simply open the software, select "File", then "Open", and import the particle labels output from example 1.2.2. Make sure to select the "Global CSV Reader".

A menu will appear on the left-hand side of the screen. Simply click "Apply" to the default information. Once the software is done loading the information, the "SpreadSheetViewer" will appear on the screen, but it can be closed right away. Select the loaded file on the pipeline browser on the left-hand side of the screen, then select the "Filters" tab at the very top of the page, search for "TableToPoints" filter and select it.

New options open up on the left-hand side menu. Under the column, select "x" for the X column, "y" for the Y column, and "z" for the Z column. Hit "Apply". This will generate a rendering of the 3D structure, but it is still hard to identify any features or read the contours, thus several modifications are needed at this step.

On the left-hand side menu, click on the cog next to the search bar, and it will "toggle advanced options". Make sure those options are available. Under "Representation", change from "Surface" to "3D Glyphs". Right below it, on the "Coloring" section, change the options from "solid" to "R". Paraview will automatically use the radius labels "R" to apply a colormap to the surface.

Scroll down to "Glyph Parameters", under "Scale Array", make sure "None" is selected. Change "Glyph Type" from "Arrow" to "Box". At this stage, a good visualization is already obtained.

Some other optional steps for improving the quality of the output include changing the background to a solid color with better contrast, adjusting the colorbar settings, and adjusting the axis and axis labels.

By following these steps (including the optional ones), we arrive at figure 1.6.

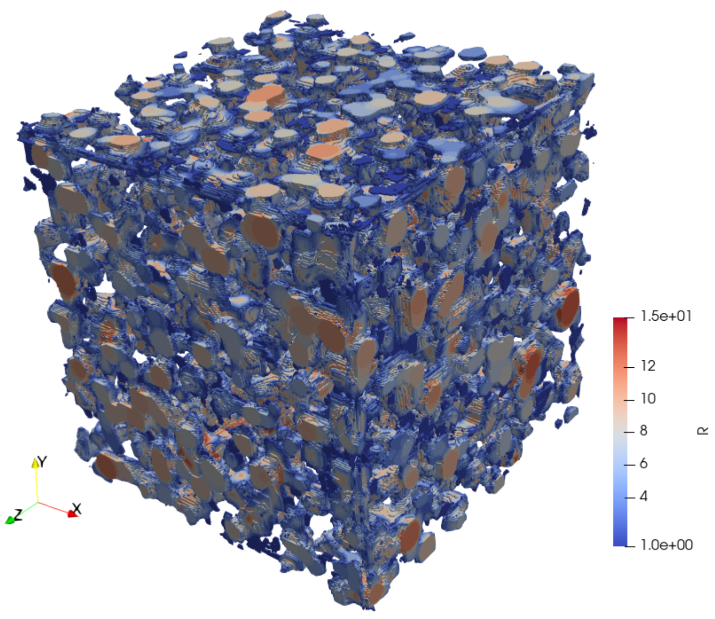


Figure 1.6: Example contour plot from processed ED-cPSD output for a 3D structure.

## 1.4 Command Line Operation

As is explained in chapter 2, there is a flexibility for the code to run in both the GUI and in a standalone command line. The command line mode is much more efficient, and enables researchers to use this same algorithm on HPC environments, where often times a GUI is not available.

This can be easily done with this project. The source files that must be included for command line operation are available on GitHub under the "src" folder. Simply add those files to the project folder, and compile the code as follows (assuming a Linux environment):

```
g++ -fopenmp ./ED_PSD_CPU.cpp -o ./ExecutableName.out
```

This will create an executable named `ExecutableName.out`. This step only needs to be done once, and after this all simulation parameters can be controlled from `input.txt`. Note that the input file can still be generated using the GUI, and then just be transferred to the HPC environment. Modifying the input text file via a bash script is also the easiest way to run large batches of simulations automatically.

# Chapter 2

# Algorithm Details

## 2.1 Motivation

The interplay between different phases in heterogeneous materials [6] is of paramount importance in determining the bulk-scale properties of said materials. More specifically, in porous media, the interplay between the particle and pore spaces are critical in determining the transport properties, as has been thoroughly discussed in scattered studies[7, 8, 9, 10, 11, 12, 13].

Several software packages and algorithms already aim take aim at reconstructing, analyzing, and characterizing the morphology of such complex systems, such as MCRpy [14] and MATBOX [15]. On the other hand, there are different interpretations on the meaning and accuracy of physical descriptors.

One such property is the particle/pore/phase size distribution, which is often reported as just the average pore/particle diameter, $D_{50}$. This is obviously an issue, as there is often no definition of what consists a particle or a pore, much less a single determination of what "size" means without context (i.e. what is the size of an ellipsoid? The semi-major axis, the semi-minor axis, the average diameter, the length, or an average of all of those?).

In some materials, like carbon-based materials for battery electrodes or some types of sandstone can be observed under scanning electron microscopy (SEM) to be largely composed of small approximately spherical individual particles. In such cases, segmenting particles within the domain and assigning them sizes is feasible, in what is considered a discrete particle-size distribution (dPSD) algorithm.

While different dPSD algorithms have different approaches, the basic scheme is the same: each domain is segmented into particles, and each particle is labeled separately. Then, the morphology of each particle is quantified, and from there the averages can be obtained (i.e. PSD and $D_{50}$. More details and a brief compilation of such algorithms can be found in Usseglio-Viretta (2020) [4].

Another avenue for obtaining the PSD is via continuous pore-size distribution (cPSD) algorithms. This class of algorithms implies a continuous assessment of the size of each phase space. In other words, no segmentation is needed and there is a continuous measurement of phase-space morphology. This interpretation is more consistent when considering materials like stochastic aluminum foams [16], where both the aluminum and pore-space are continuous and amorphous, hence incompatible with dPSD.

Even in better structured materials like triply-periodic minimal surfaces (TPMS), the cPSD measurements of particles and pore size distributions are physically interpretable, representing the probability density of thickness distribution and pore-space minimum radius distribution, respectively. Or, in the aforementioned examples of carbon-based battery electrodes and sandstone, the pore-space is simply the interstitial space defined by the absence of solid particles, hence it is also continuous and amorphous. Approaching such cases with a dPSD algorithm can be tricky, as it can quickly lead to over-segmentation [4].

Furthermore, the cPSD class of algorithms more closely resemble mercury intrusion porosimetry [3] or nitrogen desorption porosimetry [17], which are experimental methods for determining pore-size distributions. Again, a brief review and further discussion can be found in Usseglio-Viretta (2020) [4].

On the other hand, the cPSD algorithms do have several downsides [4], including the necessity for data fitting, strong assumptions about particle shape (the spherical-particle assumption), the lack of information about the location and anisotropy of the related spaces, no mechanism for particle identification, requiring large representative volumes for accurate assessment, and often times underestimation of average phase sizes.

## 2.2 cPSD via Sequential Erosion-Dilation: ED-cPSD

From any base 2D or 3D digital structure, the domain may be represented as a binary boolean array, $b$. In $b$, a true entry represents the phase of interest, and false represents everything else. In the case of pore-size distribution, the pore space is considered true, while the particle space is false. For particle-size distribution, the opposite is true. Note that this algorithm is flexible for measuring the phase-size distribution of multiple solid phase, where the phase of interest will be set as true, and everything else will be false.

The algorithm begins by expanding the false regions of the array $b$. An initial radius $r = 1$ converts every true within a radius $r$ of an interface to false. The results are then stored in array $D$. The second stage of the algorithm will shrink every false region within a radius $r$ of a true in array $D$, and store the result in array $E$. At the end of each iteration, the total number of true entries in arrays $b$, $D$, and $E$ are stored, and $r$ is increased by one.

From this basic algorithm, it is obvious that any region in $b$ that has a diameter less than $2r$ will cease to exist in $D$. If a region is not completely erased at a radius $r$ in the transformation from $b$ to $D$, then it will return after the operation from $D$ to $E$ to the exact same shape as it was in $b$. At each $r$, there is a new $E_r$, beginning at $E_0 = b$. Comparing $E_r$ to $E_{r-1}$ will show all the regions that are removed from $E$ at $r$, but not at $r - 1$. Equation (2.1) below describes how a cPSD distribution, $\Phi(r)$, can be obtained from the combination of $E_r$, $E_{r-1}$ and $b$.

$$\Phi(r) = \frac{\sum(E_r - E_{r-1})}{\sum b} \tag{2.1}$$

From an implementation perspective, note that it suffices to store one integer array $e_r = \sum E_r$, instead of storing $E_r$ at every $r$. The distribution is a probability distribution, such that $\sum_{r=1}^{r_{\max}} \Phi(r) = 1$, and $r_{\max}$ is the radius at which $\sum E_{r_{\max}} = 0$.

This algorithm is easily interpretable visually and physically, as is shown in figure 2.1.

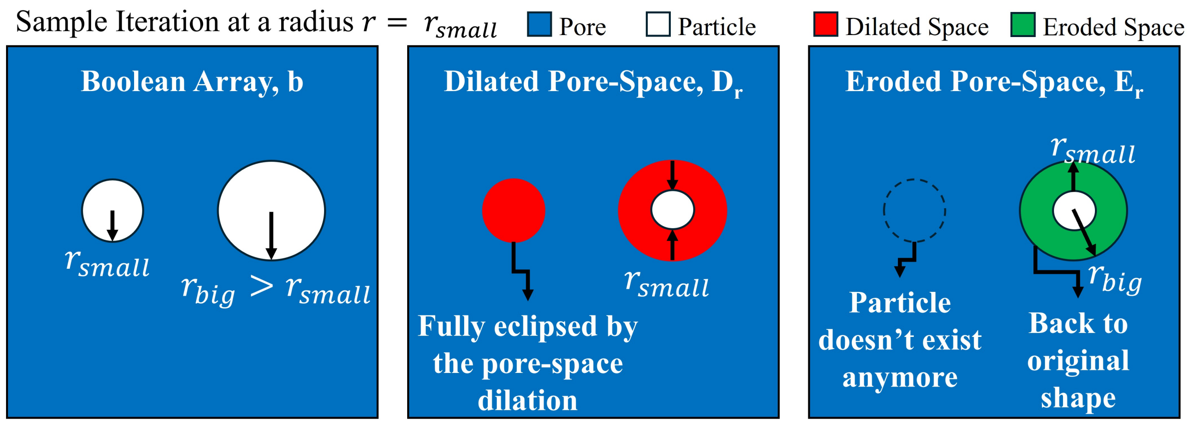


Figure 2.1: Example schematic representing the logic behind the computation of particle size distribution for a domain containing two spherical particles of different radii.

## 2.3 Erosion-Dilation Implementation

In a previous study [2], the exact algorithm described in Section 2.2 was employed for calculation PSD of large 3D structures in the scale of millions of voxels. In that study, however, the erosion and dilation operations were calculated naively from each pixel.

In other words, at each iteration, the interface between true and false along the entire $b$ array was mapped, and at each pixel in the interface, a radius $r$ must be scanned to perform the dilation. Then, the same process is repeated for $D$ to generate $E$. Therefore, at each $r$, the algorithm has a computational complexity of $O\left(Nr^3\right)$, where $N$ is the total number of voxels in the domain. This becomes a problem in domains with large pores/particles, as the algorithm scales poorly with $r$.

In this study, an alternative is proposed by calculating the exact Euclidean distance transform (EDT), and using the EDT to perform erosion and dilation operations. Evidence [18] suggests that the fastest way to calculate the exact EDT is via the Meijster algorithm [19], which is also referred as the Felzenszwalb algorithm [20]. Both authors arrived at an equivalent solution independently, but since Meijster was published earlier, this work will refer to the algorithm as Meijster's algorithm.

The Meijster algorithm calculates the EDT using four linear passes in 2D and six linear passes in 3D, therefore the algorithm is classified as linear in complexity, $O(N)$. As the studies related to porous media can reach the order of trillions of voxels [21], utilizing an algorithm with linear time complexity is quintessential, and this can be considered a major upgrade over the naive approach [2].

Both Meijster [19] and Felzenszwalb [20] claim the algorithm is scalable to an arbitrary number of dimensions, but neither provide a detailed description of the process for calculating EDT in 3D. In 2D, the EDT can be calculated as follows:

$$\text{EDT2D}(x, y) = \min\left(i : 0 \le i < m : (x-i)^2 + G(i, y)^2\right) \tag{2.2}$$

where $G(i, y)^2 = \min(j : 0 \le j < n \wedge b[i, j] : |y - j|)$. In 2D, the Meijster algorithm has two phases. In the first phase, $G(x, y)$ will be calculated for each column (fixed $x = i$). In the second phase, for a fixed row $y$, the EDT2D will be calculated according to $G(i, y)$ and equation (2.2).

To adapt the algorithm for use in 3D, only one additional phase is necessary. In the first phase, $G(x, y, z)$ is calculated in the exact same manner, for each column in the entire domain (fixed $x$ and $z$). Phase 2 is also the exact same, in which EDT2D is calculated for each row (fixed $y$ and $z$). Phase 3 is a repeat of phase 2, with the only change being that EDT3D depends on EDT2D, and not $G(x, y, z)$. For each slice (fixed $x$ and $y$), EDT3D is calculated according to equation (2.3).

$$\text{EDT3D}(x, y, z) = \min\left(k : 0 \le k < p : (z-k)^2 + EDT2D(x, y, k)\right) \tag{2.3}$$

Beyond reducing the time complexity of the algorithm, calculating the EDT before erosion or dilation has an additional benefit. In the algorithm presented in Section 2.2, an EDT in the true phase is necessary to transform from $b$ to $D_r$. The transformation follows the simple rule that if $\text{EDT3D}(x, y, z)\,|_{b=1} \le r^2$, then $D(x, y, z)_r = 0$. Whether $\text{EDT3D}|_{b=1}$ is calculated from $b$ or $E_{r-1}$, the dilation yields the same $D_r$. Therefore, $\text{EDT3D}|_{b=1}$ only needs to be calculated once in the entire algorithm.

The EDT for the false phase of $D$ is necessary to transform $D_r$ into $E_r$ at each iterative step. In other words, for each $r$, one EDT3D must be calculated using the Meijster algorithm in 3D. Since this is by far the most time consuming operation, each iteration of the successive dilation-erosion algorithm takes approximately the time it takes to create one EDT map.

Finally, each phase of the Meijster algorithm can be thought of as having a set of independent operations, making it an embarrassingly parallel algorithm [22]. That means that on a cluster with $k$ threads (independent of being on CPU or GPU), the time complexity of generating the EDT should be at or near $O\left(\frac{N}{k}\right)$. This type of algorithm scales extremely well with additional computational resources in HPC environments.

## 2.4 Phase Labeling

Most cPSD algorithms do not have any mechanisms to backtrack a phase segmentation from the calculation of phase-size distribution [4]. This can be considered as one of the downsides of cPSD algorithms, as there is no mechanism to retrieve particle morphology or inhomogeneity from the cPSD metric.

This is not the case for the ED-cPSD. In fact, the base software produces two different segmentation metrics. One is displayed in sections 1.3.2 and 1.3.3, and that is the labeling via radius of removal.

This labeling mechanism based on radius emerges directly from the algorithm discussed in section 2.2. An array $R$ is initialized at the beginning of the phase-size distribution, and it is all initialized to zeroes. Then, at any given radius, $r$, the array $E_r$ is compared to the original boolean array, $b$. If $b|_{i,j,k} == 1$, $E_r|_{i,j,k} == 0$, and $R|_{i,j,k} == -1$, the $R|_{i,j,k} = r$.

In simpler terms, array $R$ keeps track of which $r$ eliminated pixel coordinates $(i, j, k)$. In order to further group pixels by radius of removal, another step is implemented is necessary to label clusters of pixels with the same radius at the same location.

A flood-fill algorithm `https://en.wikipedia.org/wiki/Flood_fill` is applied at the pixel level for this task, grouping clustered pixels with the same $r$. This is how labels $L$ are derived. This process is done by ascending $r$, as if it was done by descending $r$, it would be analogous to a Watershed algorithm [23, 21], albeit at a lower resolution and with no segmentation post-processing.

## 2.5 GUI and the Qt Framework

The Qt framework `https://www.qt.io/` is to wrap the C++ code into a GUI. The publication outlines a flexible double computation model that allows the same functions to be called from inside the GUI or directly from the command line.

The GUI serves a dual purpose: it can be used to aid in the creation of input files and it can also be used to run the code. The creation of input files is relatively straight forward, and it is just a matter of connecting the buttons to the desired outputs using a signals and slots built in to Qt, and therefore this portion is not covered in this document.

However, running the code inside the GUI is more complex, and deserves some attention. Understanding this complex interplay of the C++ execution, the Qt framework, the GUI execution loop, the worker thread responsible for the simulation, and the additional threads recruited using OpenMP for the parallel execution of the code are fundamental in understanding certain decisions regarding the code structures.

The biggest challenge in running a resource intensive simulation as part of a GUI is the following: the GUI execution loop must be on constantly, therefore the simulation can't be run within the execution loop, as that will freeze the GUI and cause the program to crash.

Several methods were considered to avoid this issue. Without going into detail about the methods that didn't work, the basic idea is that there is also a fine line between independent execution where the GUI execution loop and the code execution can still communicate with each other, but are independent enough that the code execution does not cause the GUI to freeze. Methods like Qt::Concurrent allow for the independent execution, but lack in the communication department.

The method used is to separate the GUI execution loop using a class definition for the main window, and a class definition for the worker thread. The worker has an execution loop that is completely outside of the execution loop of the main window, and they communicate via some shared variables locked by a mutex, and also the worker is able to send messages to the main window via signals. This computational model is best exemplified by looking at the Mandelbrot example from the Qt documentation, which can be found here: `https://doc.qt.io/qt-6/qtcore-threads-mandelbrot-example.html`.

## 2.6 Modifying Base Algorithm

This section refers to the `ED_cPSD_CPU.cpp` and `.hpp` files included in the `src` folder on GitHub. These files are only used for command line operation, and the files used for the GUI operation are in a different folder (covered in section 2.7).

The structure of the code is fairly simple: the main `.cpp` file is only used to read the input file, store the information from the input file in the related data structures, and then decide between 2D and 3D operation based on the user input.

The `.hpp` file contains everything else. The code is not lengthy, hence it was more convenient to have all functions and data structures in the same file. There are three essential data structures. The struct *options* handles all of the user entered options. The structs *sizeInfo* and *sizeInfo2D* handle mesh parameters from 3D and 2D simulation modes, respectively.

The functions that actually control the simulations are called `Sim2D` and `Sim3D`, and they both take a pointer to *options* as input. In structure, these functions are very similar: they read the input files, declare necessary structure arrays, and call other functions for running pore and/or particle size distributions according to the user input.

All functions have a header explaining what the expected inputs and outputs are, as well as explaining what the function actually does. Overall the code is short and full of comments, so modification should be easy. Below is an example of function declaration and comment header:

```
int partSD_2D(options *opts,
              sizeInfo2D *info,
              char *P,
              char POI)
{
    /*
        Function partSD_2D:
        Inputs:
            - pointer to options struct
            - pointer to structure info struct
            - pointer to phase-array
            - char phase of interest
        Outputs:
            - None.
        Function will calculate particle size distribution of array P.
    */

    ...
```

## 2.7 Modifying GUI Code

The GUI code follows a lot of the same conventions already covered in section 2.6. However, what is very different is the structure of the files, as there are a lot more files in this mode of operation. Additionally, several functions are re-formatted to below to a class, and albeit they behave the same, the implementation is slightly different.

This version of the code has many more files than the simple command line version of the code. To begin with, the project is structured around the `CMakeLists.txt` file, which is the way Qt and CMAKE are able to organize all of the files together into one cohesive executable. The structure of the `CMakeLists.txt` file won't be covered.

There is one singular file with a `.ui` extension. This is the Qt user interface file, and can be opened via the Qt Designer or via the Qt Creator "Design" mode. This file contains all information about the location of everything in the GUI.

Then, there are two source files: `main.cpp` and `mainwindow.cpp`. The first file is only start the QApplication and begin the execution loop of the main window. The second file actually contains all of the functions, slots, and signals connections, as well as class declarations. To understand those, we must first understand the helper files structure.

There are four helper files in total. The `data_structs.hpp` file only contains the data structure definitions, identical to the definitions in the command line version. This is mainly because the classes can inherit data structures or use them as input for protected functions. Leaving the data structures definition in the `ED_PSD_CPU.hpp` would create issues with recursive definitions.

File `stb_image.h` is straight from `https://github.com/nothings/stb`, without any modifications. This is used for reading the `.jpg` files.

The file `main_window.h` has the class definitions for the two classes in this work. The *MainWindow* class, as the name suggests, holds all of the information and adds functionality to all of the buttons in the main window. This class is the backbone of the GUI, and is also used to invoke the *Worker* class.

The *Worker* class is what controls the simulation, and it is a re-implementation of the QThread class. This runs mostly in the background, but it also handles the communication between the two classes via signals. A mutex is also common to both *Worker* and *MainWindow*, and is implemented in such a way that *MainWindow* can pass essential information to *Worker*, such as orders to abort the simulation or close the program.

The simulation functions from `ED_PSD_CPU.hpp` are re-implemented as functions in the *Worker* class. These functions are refactored such that the output is sent as a signal to the main window, as opposed to printing in the command line. Auxiliary functions, such as reading images or calculating EDT are left in the `ED_PSD_CPU.hpp`, as those functions don't require much communication or are executed quickly, hence they don't need to be part of the *Worker* class.

The function headers follow a similar convention to what is defined in section 1.4, thus making the code readable and easy to modify. If there are any further questions about the implementation, don't hesitate to reach one of the authors.

Finally, the `resources` folder contains a `.ico` file, which is used as the icon for the application. If all of these files are present in the computer, and Qt is added to the path, then the project should be easy to build with CMAKE. However, if Qt is not on the path, either use the static version of the project available on GitHub with manually linked libraries, or consider using the Qt Creator (Qt's own IDE) to build the project. For a new user, installing Qt and using the Qt Creator is highly recommended.

# Chapter 3

# Case Studies

## 3.1 Case Study 1: NREL Battery Microstructure Dataset

**This is the same case study as shown in the publication. It will just be added here once it is done there.**

# Chapter 4

# Additional Information

## 4.1 Acknowledgments

This work used Expanse(GPU)/Bridges2(CPU) at SDSC/PSC through allocations MAT210014 and MAT230071 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants 2138259, 2138286, 2138307, 2137603, and 2138296 [24].

X.L. highly appreciates the support from the National Science Foundation (Award 1941083 and 2329821).

The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number 18/CJ000/08/08.

## 4.2 License

This code is protected by the MIT license. For more information, refer to the "LICENSE.txt" file. For information on why licensing is important, check out the GitHub page on licensing: `https://tinyurl.com/3bctstmc`.

The basic gist of it is, if you use the code, please acknowledge the developers via citing one of the publications, and do not claim the work as your own.

# Bibliography

[1] Schneider, C. A., Rasband, W. S. & Eliceiri, K. W. Nih image to imagej: 25 years of image analysis. *Nature Methods* **9**, 671–675 (2012). URL http://dx.doi.org/10.1038/nmeth.2089.

[2] Adam, A., Wang, F. & Li, X. Efficient reconstruction and validation of heterogeneous microstructures for energy applications. *International Journal of Energy Research* (2022). URL https://onlinelibrary.wiley.com/doi/full/10.1002/er.8578.

[3] Münch, B. & Holzer, L. Contradicting geometrical concepts in pore size analysis attained with electron microscopy and mercury intrusion. *Journal of the American Ceramic Society* **91**, 4059–4067 (2008).

[4] Usseglio-Viretta, F. L. E. *et al.* Quantitative relationships between pore tortuosity, pore topology, and solid particle morphology using a novel discrete particle size algorithm. *Journal of The Electrochemical Society* **167**, 100513 (2020). URL https://iopscience.iop.org/article/10.1149/1945-7111/ab913bhttps://iopscience.iop.org/article/10.1149/1945-7111/ab913b/meta.

[5] Ahrens, J. P., Geveci, B. & Law, C. C. Paraview: An end-user tool for large-data visualization. In *The Visualization Handbook* (2005). URL https://api.semanticscholar.org/CorpusID:56558637.

[6] Wu, X. & Zhu, Y. Heterogeneous materials: a new class of materials with unprecedented mechanical properties. *Materials Research Letters* **5**, 527–532 (2017).

[7] Lin, G. *et al.* Effect of pore size distribution in the gas diffusion layer adjusted by composite carbon black on fuel cell performance. *International Journal of Energy Research* **45**, 7689–7702 (2021). URL https://onlinelibrary.wiley.com/doi/full/10.1002/er.6350https://onlinelibrary.wiley.com/doi/abs/10.1002/er.6350https://onlinelibrary.wiley.com/doi/10.1002/er.6350.

[8] Cui, C. L., Schweich, D. & Villermaux, J. Influence of pore diameter distribution on the determination of effective diffusivity in porous particles. *Chemical Engineering and Processing* **26**, 121–126 (1989).

[9] Liu, X. *et al.* The influence of pore size distribution on thermal conductivity, permeability, and phase change behavior of hierarchical porous materials. *Science China Technological Sciences* **64**, 2485–2494 (2021). URL http://dx.doi.org/10.1007/s11431-021-1813-0.

[10] Adam, A., Fang, H. & Li, X. Effective thermal conductivity estimation using a convolutional neural network and its application in topology optimization. *Energy and AI* **15**, 100310 (2024). URL http://dx.doi.org/10.1016/j.egyai.2023.100310.

[11] Li, J. X., Rezaee, R., Müller, T. M. & Sarmadivaleh, M. Pore Size Distribution Controls Dynamic Permeability. *Geophysical Research Letters* **48**, e2020GL090558 (2021). URL https://onlinelibrary.wiley.com/doi/full/10.1029/2020GL090558https://onlinelibrary.wiley.com/doi/abs/10.1029/2020GL090558https://agupubs.onlinelibrary.wiley.com/doi/10.1029/2020GL090558.

[12] Tian, S., Ren, W., Li, G., Yang, R. & Wang, T. A Theoretical Analysis of Pore Size Distribution Effects on Shale Apparent Permeability. *Geofluids* (2017). URL https://doi.org/10.1155/2017/7492328.

[13] Kapat, K. *et al.* Influence of Porosity and Pore-Size Distribution in Ti 6 Al 4 V Foam on Physicomechanical Properties, Osteogenesis, and Quantitative Validation of Bone Ingrowth by Micro-Computed Tomography. *ACS Appl. Mater. Interfaces* (2017). URL `www.acsami.org`.

[14] Seibert, P., Raßloff, A., Kalina, K., Ambati, M. & Kästner, M. Microstructure characterization and reconstruction in python: Mcrpy. *Integrating Materials and Manufacturing Innovation* **11**, 450–466 (2022). URL `https://doi.org/10.1007/s40192-022-00273-4`.

[15] Usseglio-Viretta, F. L. *et al.* Matbox: An open-source microstructure analysis toolbox for microstructure generation, segmentation, characterization, visualization, correlation, and meshing. *SoftwareX* **17** (2022).

[16] Stallard, S., Jiang, H., Chen, Y., Bergman, T. L. & Li, X. Exploring the design space of the effective thermal conductivity , permeability , and stiffness of high-porosity foams. *Materials & Design* **231**, 112027 (2023). URL `https://doi.org/10.1016/j.matdes.2023.112027`.

[17] Joyner, L. G., Barrett, E. P. & Skold, R. The determination of pore volume and area distributions in porous substances. ii. comparison between nitrogen isotherm and mercury porosimeter methods. *Journal of the American Chemical Society* **73**, 3155–3158 (1951).

[18] Fabbri, R., Da F. Costa, L., Torelli, J. C. & Bruno, O. M. 2D Euclidean distance transform algorithms: A comparative survey. *ACM Computing Surveys* **40** (2008).

[19] Meijster, A., Roerdink, J. B. T. M. & Hesselink, W. H. *A General Algorithm for Computing Distance Transforms in Linear Time*, 331–340 (Springer US, Boston, MA, 2000). URL `https://doi.org/10.1007/0-306-47025-X_36`.

[20] Felzenszwalb, P. F. & Huttenlocher, D. P. Distance transforms of sampled functions. *Cornell Computing and Information Science Technical Report TR20041963* **4**, 1–15 (2004). URL `https://theoryofcomputing.org/articles/v008a019http://ecommons.cornell.edu/handle/1813/5663%5Cnhttp://ecommons.library.cornell.edu/handle/1813/5663`.

[21] Barnes, R. Parallel priority-flood depression filling for trillion cell digital elevation models on desktops or clusters. *Computers and Geosciences* **96**, 56–68 (2016).

[22] Wilkinson, B. & Allen, M. C. *Parallel Programming: Techniques and applications using networked workstations and parallel computers* (Pearson/Prentice Hall, 2005), 2 edn.

[23] Barnes, R., Lehman, C. & Mulla, D. Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models. *Computers and Geosciences* **62**, 117–127 (2014).

[24] Boerner, T. J., Deems, S., Furlani, T. R., Knuth, S. L. & Towns, J. Access: Advancing innovation: Nsf's advanced cyberinfrastructure coordination ecosystem: Services & support. In *Practice and Experience in Advanced Research Computing*, PEARC '23, 173–176 (ACM, 2023). URL `http://dx.doi.org/10.1145/3569951.3597559`.